DMS
Program Transformations for Practical Scalable Software Evolution

Ira Baxter    Chris Pidgeon    Michael Mehlich
Semantic Designs, Inc.
www.semanticdesigns.com

May 2004
Software Change Tools Need Infrastructure Too

• To build a conventional application
  – Define specifications
  – Implement application
  – Use Operating System to provide standard services
    • File/Terminal I/O, CPU management, Security

• To build an automated software change tool
  – Define specifications
  – Implement tool
  – Use …?… to provide standard services
    • Lexing/Parsing
    • Life after Parsing: Tree/Symbol table building,
      Analysis management, Transformation, PrettyPrinting

• This talk: a practical change-tool infrastructure
Transformation Systems

Stepwise Conversion of Specs to Code

Rqmts \( \rightarrow \) Spec \( f_s \) \( \rightarrow \) Transform Engine \( \rightarrow \) Prog \( f_G \)

Transforms

distributive law \( t_1 \)
unity multiplier \( t_2 \)
remove parentheses \( t_{k-2} \)
like-term combination \( t_{k-1} \)
factoring \( t_k \)

\[ f_s \rightarrow f_1 \rightarrow \ldots \rightarrow f_{k-1} \rightarrow f_k \rightarrow f_G \]

\[ (x-1)y+2y \rightarrow (xy-1y)+2y \rightarrow xy-y+2y \rightarrow xy+y \rightarrow (x+1)y \]
Practical Transformation Machinery

- Theory long established
  - Source to Source transforms ("rewrites")
  - Refinement theory
- Emphasis now focused on *change*
- Need integrated mechanisms
  - Parsing, PrettyPrinting
  - Name/Type Resolution, Rewriting, Analyzers
  - Essential technology already developed
- *Practical tools need support for scale*
  - Large real codes
  - Multiple languages
  - Symbolic Computation for analysis and change
The Design Maintenance System Vision

- Transformational Design
  - Functionality Spec \((f_0)\) + Performance Spec \((G_{rest})\) + Derivation + Justification + Alternatives
  - Metaprogram driven automation

- Incremental Updates as \(\Delta s\)
  - Specification, Performance, Technology \(\Delta s\)
  - \(\Delta s\) drive design revision:
    - retain transforms that commute with delta

\[\Delta @ p (C_i @ q(f_i)) = C_i @ q'(\Delta' @ p'(f_i))\]
DMS Software Reengineering Toolkit

- **Automated** source code analysis and **modification**
  - *Leverage transformation machinery needed to build DMS vision*

- Enables wide variety of SE tasks to be automated
  - **Commercial applications**
    - Source Formatters, Hyperlinked Source Browsers
    - Documentation extraction
    - Metrics
    - Preprocessor conditional simplification
    - Test Coverage and Profiling tools
    - Clone Detection and removal
    - XML DTD compilation
    - DSL code generation: Factory Automation
    - Migrations (JOVIAL to C)
  - **Research applications**
    - Aspect-weaving (U. Alabama Birmingham)
    - Large-scale C++ component restructuring (SD/Boeing)
    - Code generation/quality checking for spacecraft (NASA/JPL)
DMS Implementation
"Software Reengineering Toolkit"

Coded in PARLANSE
DMS Toolkit = Generalized Compiler

• Underlying Hypergraph representation: trees, graphs, …
• Parsing/Prettyprinting
  – UNICODE lexer with binary conversions, lexical format/comment capture
  – GLR (context-free) parser with automatic tree builder
  – Prettyprinting by "Text Box" building language
• Analysis
  – Multipass attribute grammars
  – Generalized symbol table support: inheritance, overloading, …
  – Next: Generic Control/Data Flow
• Transformation
  – Complete procedural AST interface => procedural transforms (& analyzers)
  – Conditional Source to Source transforms w/ associative/commutative laws
  – Next: Goal-directed metaprogramming
• Predefined Domains
  – Specification, Technology, and Implementation languages
    Spectrum, .MDL XML, SQL, IDL C/C++, C#, Java, COBOL, …
Fundamental Issue: *Scale*

- Engineering hard but straightforward
  - Ugly details: C preprocessors, etc.
- Reasoning/Analysis costs
  - "Incremental" Knowledge capture => domains
  - Computers do Symbolic computation slowly => Parallel foundations
  - Future: RuleSet compilers
- Legacy Systems are huge
  - MSLOC + tens of thousands of files
    - Careful design of hypergraph to conserve space
  - Arbitrary languages => robust parsing technology
    - Need for domain agility: fast domain/dialect definition
      * Use domain notations define knowledge*
  - Applications use multiple languages
    - reasoning and transforms must work with mix
- Other scale issues
  - Software Versions, Large Engineering Teams, Long term Transactions
Knowledge Capture:

DMS Domain Parts

- **Syntax** (domain notation)
  - External Form (what you can say: string or graphical)
  - Internal Form (How DMS stores it)
  - Parser (how to convert external form to internal form)
  - PrettyPrinter (how to display the Internal Form)

- **Semantics** (what the domain *means*)
  - Analyzers (how to analyze in the domain)
  - Optimizations (how to optimize in the domain)
  - Refinements (how to transform one domain to another)
  - (Attachments) (procedures to enhance DMS efficiency)
A Domain Network

Specific Applications
- Natural Language Tellers
- Electronic Funds Transfer
- Punch Press Control
- Fighter Aircraft Navigation

Generic Applications
- Natural Language Parsing
- Money Management
- Real Time Control
- Global Navigation

Computer Science
- Graphical User Interface
- Data Structures
- Parallelism / Distributed Computation

Execution Model
- OOP
- Logic
- Functional
- Data Flow

Target Execution
- C++
- Prolog
- Haskell
- Occam

refines to Natural Language Teller

parcels to

parallelism / distributed computation

optimize
DMS Domain for Java
Parser + Pretty Printer

```
nested_class_declaration = nested_classModifiers class_header class_body;
  <<PrettyPrinter>>: { V(H(nested_classModifiers,class_header),class_body); }

class_header = 'class' IDENTIFIER;
  <<PrettyPrinter>>: { H('class',IDENTIFIER); }
class_header = 'class' IDENTIFIER 'implements' name_list;
  <<PrettyPrinter>>: { H('class',IDENTIFIER,'implements',name_list); }

class_header = 'class' IDENTIFIER 'extends' name;
  <<PrettyPrinter>>: { H('class',IDENTIFIER,'extends',name); }
class_header = 'class' IDENTIFIER 'extends' name 'implements' name_list;
  <<PrettyPrinter>>: { H('class',IDENTIFIER,'extends',name,'implements',name_list); }

class_body = '{' class_body_declarations '}'
  <<PrettyPrinter>>: { V(H('{',STRING(" "),class_body_declarations),'')); }

nested_classModifiers = nested_classModifiers nested_classModifier;
  <<PrettyPrinter>>: { H(CH(nested_classModifiers[1]),nested_classModifier); }

... + 300 more rules...(COBOL is 3500!)```
Parsing to Abstract Syntax Trees
A Program Representation analyzable by Computers

• Use DMS grammar domain to define language syntax
• DMS generates lexer/parser automatically
• Reengineering parser reads source file(s)
  – Carries out lexical conversions (e.g., FP text -> IEEE binary fp)
  – Captures comments and formats of literals
  – Builds Abstract Syntax Tree
  – Records Position of every node (file, line, col)

• Present capability for the following domains
  – Specification: Spectrum, BNF, Rose Models
  – Technology: XML, IDL, SQL
  – Implementation: C/C++, C#, COBOL, Java, Ada, VB6, Fortran, Verilog
#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/Xkeysym.h>

double L, o, P, J, K, W, X, S, [999], Z, H, E, N, M, N, j, d = dt, T, z, u, y, w, s = 74.5, l = 221, x = 17.26, a, B, A = 32.2, c, f, h;

int W, N, C, y, p, U;

Window z; char f[52];

CC k; main()
{
    Display e = XOpenDisplay(0);
    z = RootWindow(e, 0);
    for (XSetForeground(e, k = XCreateGC(e, z, 0), BlackPixel(e, 0)) ;
        scanf("%lf%lf%lf", y, n, w, y)+1; y += j)
        XSelectInput(e, z = XCreateSimpleWindow(e, z, 0, 0, 400, 400, 0, 0, WhitePixel(e, 0), KeyPressMask));
    for (XMapWindow(e, z); T = sin(o) ;
        struct timeval G = { 0, dt * 1e6 }
    )
        K = cos(j); N = 1e4; M += N; Z = D * K; F += P; t = E * K; N = 0; f = 0; D += DT; K += J; U += U;
    XClearWindow(e, z); T += T*K + D*B;} = 3; D = 2*K * E * T + D; for (a += sin(I * DT * X * E * T);
        t = T * (x * t + P * M * x - T * D); if (p[n]*w[p] < 0)
        N = 1e4;
    else if (q / K * 4 * E * 2 + 2 / K * D * a * E * T)
        XDrawLine(e, z, k, N, U, C); N = q; q = N; U = C;
    ++p
    L = + = (K * T + P * M * a); T = X * X + 1 * I * M * N;
    XDrawString(e, z, k, 20, 380, f, 17); d += v / l + 15; i += (B * I - T + X * E)
    for (; XPending(e); u *= C)
        XNextEvent(e, 42);
    ++(*XLookupKeysym(42, xkey, 0)) - IT?
    DLT UP-N F 4 &
    J: k u: kh)
    --*{DN - H? N - D? TN =
    RT7u: & W: k: 0 Z[r + 1]
    } ; ; m = 15 * F/l ;
    c = (1 / N / 1, 1 * K
    + I * M * a) * X
    H
    = 2 * (x + X * F + 1 +
       -END / 4 * 9 / l +
    t * 3 / 2 / T)
    / 0 /
    K = E * M + (h * 1e4 / I + (F * 0.5 + E) * F)
    / X * d - B * A
    ; a = 0.6 / 1 * d
    X += (d * 1.1 / 2
    = (1.9 * E + a
    + 0.6 * x) / 1e3
    - M + v * A
    )
    i +=
    K
    . W; d;
    sprintf(f,
        "%5d %5d %7d",
        p = l
    / 1.7, C = 3E +
    O * 57.345500, (int) i;
    t = T * (0.45 - 14 / 27 * x * a - 130 - J * 14) * 1218 / v
    / 2312 select (p = 0, 0.6, 0.4, 0.7)
    * 6.3 + M * 3.85 + B * E - 0.14 - 1.4
    ) / 1072 * _ + D = cos(o)) 0 = sin(o); f)
}
Analyzers using Attribute Evaluation

Analyzing program properties using tree structure

```java
/* Fib.java */

public class NumberTheory {
    int Fib(int x) {
        if (x < 1) return 1; // base case
        else return Fib(x-1) + Fib(x-2);  
    }
}
```

Not shown: File/line/column annotation on each node
AttributeEvaluatorSpecification

determining Constants used

ATTRIBUTES { SetOfReal Constants; } 
Statement = 'return' Expression ;
<<UseConstants>> { Statement.Constants = Expression.Constants; } 
Expression = Sum ;
<<UseConstants>> { Expression.Constants = Sum.Constants; }

Sum = Sum '−' Term ;
<<UseConstants>> { Sum[0].Constants = UnionReals(Sum[1].Constants,Term.Constants); }
Sum = Sum '+' Term ;
<<UseConstants>> { Sum[0].Constants = UnionReals(Sum[1].Constants,Term.Constants); }
Term = '(' Expression ')' ;
<<UseConstants>> { Term.Constants = Expression.Constants ; }
Term = IDENTIFIER ;
<<UseConstants>> { Term.Constants = EmptySetOfReals(); }

Term = NATURAL ;
<<UseConstants>> { Term.Constants = SingletonRealSetFromNatural(NATURAL.); }
Term = DOUBLE ;
<<UseConstants>> { Term.Constants = SingletonRealSetFromDouble(DOUBLE.); }
Term = IDENTIFIER '(', ArgumentList ')' ;
<<UseConstants>> { Term.Constants = ArgumentList.Constants; }
ArgumentList = Expression ;
<<UseConstants>> { ArgumentList.Constants = Expression.Constants; }
ArgumentList = ArgumentList ',' Expression ;
<<UseConstants>> { ArgumentList[0].Constants = UnionReals(ArgumentList[1].Constants,
Expression.Constants); }
Symbol Table Support

• Needed to support non-context-free transformations
• = Set of Symbol Spaces
• Each Symbol Space
  – Map from identifier to arbitrary value
    • Domain specific matching used to implement overloading
  – Multiple lexical-scope parent links with integer priorities
  – Lexical-scope search mediated by attached actions
• Typically constructed by attribute evaluation
  – Fully implemented: C, C++, Java, COBOL
Optimization transform
in DMS Rewrite Rule Language

default base domain C;

rule use-auto-increment(v: lvalue):
    statement -> statement
    "v = v +1"
rewrites to
    "v++"
    if no_side_effects(v);


After:   (*Z)[a>>2]++;
Refinement transforms

*Jovial to C*

default source domain Jovial;

default target domain C;

private rule refine_data_reference_dereference_NAME
    (n1:identifier@C,n2:identifier@C)
    :data_reference->expression
  = "\n1\:NAME @ \n2\:NAME" -> "\n2->\n1".

private rule refine_for_loop_letter_2
    (lc:identifier@C,f1:expression@C,
     f2:expression@C,s:statement@C)
    :statement->statement
  = "FOR \lc\:loop_control :
     \f1\:formula BY \f2\:formula; \s\:statement"
  ->
  "\{ int \lc = (\f1);
     for(;;\lc += (\f2)) { \s } }"
  if is_letter_identifier(lc).
Refinement Transforms in Action

Jovial to C

JOVIAL Source:

```
FOR i: j*3 BY 2 ;
    x@mydata = x@mydata+I;
```

Translated C Result:

```
{ int i = j*3;
    for (; ; i+=2)
        { mydata->x = mydata->x + i}
}
```

Typically lots of small transforms for full translation

~2500 rules to translate full Jovial
A More Complex Example

Jovial to C

```c
#include "jovial.h"

static struct
{
    /* Main status boolean */
    enum { V(YES), V(NO) );
    tftp_g_twrdet_status(V(YES),V(NO));
END

TYPE TFP'D'TWRDET'TABLE TABLE (7:23) W 3;
BEGIN
    ITEM TFP'ITM S 3 POS(0,3); "cube axis"
END

%begin proc%
PROC proc'a(c1) S;
BEGIN
    ITEM match'count U 6;
        %an item%
    ITEM c1 C 5; "parameter value"
    ITEM c2 C 7;
    IF c1 <= c2 AND c2 > c1;
        match'count = UBOUND(TFP'D'TWRDET,0) +
               UBOUND(TFP'D'TWRDET'TABLE,0);
    "result off by 1 so adjust"
    match'count = match'count+1;
    BEGIN
        match'count=match'count/2;
        PROC'A = match'count; % return answer %
    END "cleanup and exit";
END "end proc"
```

packed tables with bit offsets,
typedefs, functions,
string operations, comments

Equivalent C
(used with hand-coded macro library)
PARLANSE

PARallel LANguage for Symbolic Expression

- Used to implement DMS & DMS-based applications
  - Goal: provide support expensive, large-scale symbolic computation
- Compiled, C-like programming language
  - Scalar and compound data types; pointers without casts
- Parallelism Support for SMP workstations; 2-3x speedup NOW
  - Explicitly specified parallelism
    - Dynamic forks
    - Static partially-ordered computations
    - Compiler chooses how much to implement
  - signal and wait primitives
- Software Engineering Support
  - Dynamic strings and arrays
  - Storage allocation pools with block release
  - Exception handling integrated with parallelism
  - Debug time checking: array indices, union tags, bad pointers, …

\[
\begin{align*}
  ( &; \text{first} (\ll \text{second} \text{fourth}) \\
  &; \text{second} (\leftarrow z (\text{fib} x)) \\
  &; \text{third} (\text{sort} \ (\text{.} \ y)) \\
  &; \text{fourth} (\gg \text{third}) \ (= f y:x) )\end{align*}
\]
Uses of PARLANSE in DMS

- DMS core implementation
  - 250K SLOC; SE support was essential!
  - Parallel safe data structures (e.g., symbol tables)
- Application tool "metaprogramming" language
  - Can often parallelize actions of tools at high level
- Custom domain support/escapes
  - Parallel Symbol table construction for Java (3500 files!)
  - Parallel procedural Clone analysis
- Target of DMS attribute rule compiler
  - Reliable partial-order parallel programs by construction!
    - C++ Name resolver: 40K SLOC .ATG --> 250K SLOC PARLANSE
- Future: fine-grain parallel rewriting/inference
- *Importance grows with scale of automation ambitions*
  - You can't decide to parallelize when you hit a scale barrier!
DMS: Conclusion

- Transformation technology maturing
  - Practical value *now* for
    - Massive change
    - High quality code generation

- Need generalized compiler-like infrastructure
  - Definable parsers, prettyprinters, transforms, analyzers
  - Must *scale* to application systems with MSLOCs
  - *Amortizes tool costs over many custom tools*

- Much work remaining to achieve DMS *vision*
  - Capture of transform sequence and rationale
  - Implementation of design revision
  - Transactioning to support teams of engineers